Identification of Dynamic Circuit Specialization Opportunities in RTL Code

TOM DAVIDSON, ELIAS VANSTEENKISTE, KAREL HEYSE, KAREL BRUNEEL, and DIRK STROOBANDT, Ghent University, ELIS Department

Dynamic Circuit Specialization (DCS) optimizes a Field-Programmable Gate Array (FPGA) design by assuming a set of its input signals are constant for a reasonable amount of time, leading to a smaller and faster FPGA circuit. When the signals actually change, a new circuit is loaded into the FPGA through runtime reconfiguration. The signals the design is specialized for are called parameters. For certain designs, parameters can be selected so the DCS implementation is both smaller and faster than the original implementation. However, DCS also introduces an overhead that is difficult for the designer to take into account, making it hard to determine whether a design is improved by DCS or not. This article presents extensive results on a profiling methodology that analyses Register-Transfer Level (RTL) implementations of applications to check if DCS would be beneficial. It proposes to use the functional density as a measure for the area efficiency of an implementation, as this measure contains both the overhead and the gains of a DCS implementation. The first step of the methodology is to analyse the dynamic behaviour of signals in the design, to find good parameter candidates. The overhead of DCS is highly dependent on this dynamic behaviour. A second stage calculates the functional density for each candidate and compares it to the functional density of the original design. The profiling methodology resulted in three implementations of a profiling tool, the DCS-RTL profiler. The execution time, accuracy, and the quality of each implementation is assessed based on data from 10 RTL designs. All designs, except for the two 16-bit adaptable Finite Impulse Response (FIR) filters, are analysed in 1 hour or less.

Categories and Subject Descriptors: B.5.2 [**Optimization**]: Simulation

General Terms: Field-Programmable Gate Array, Runtime Reconfiguration, Computer-Aided Design

Additional Key Words and Phrases: Dynamic circuit specialization, RTL profiling

ACM Reference Format:
Tom Davidson, Elias Vansteenkiste, Karel Heyse, Karel Bruneel, and Dirk Stroobandt. 2015. Identification of dynamic circuit specialization opportunities in RTL code. ACM Trans. Reconfig. Technol. Syst. 8, 1, Article 4 (February 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2629640

1. INTRODUCTION

In modern Field-Programmable Gate Arrays (FPGAs), it is possible to reconfigure specific configuration bits at runtime. This runtime reconfiguration can be used for Dynamic Circuit Specialization (DCS), which specializes the circuit implemented on the FPGA at runtime. In DCS, the circuit implemented on the FPGA is specialized for the specific value of a chosen set of input signals. When these signals change, the FPGA is reconfigured with a FPGA circuit specialized for the new input values. The set

This work was funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).
Authors' addresses: ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent; emails: {Tom.Davidson, Elias.Vansteenkiste, Karel.Heyse, Karel.Bruneel, Dirk.Stroobandt}@ugent.be.

DOI: http://dx.doi.org/10.1145/2629640

of input signals the FPGA circuit is specialized for are called parameters. DCS is used to build faster and smaller FPGA implementations. However, there are many possible DCS implementations of an application and not every DCS implementation will show significant gains. Moreover, not every design has a DCS implementation that improves the original implementation. In this article, we address the identification of designs that can benefit from a DCS implementation. We present a profiling methodology that takes both the overheads and the gains from DCS into account.

Determining the potential benefits of a DCS implementation is a complex problem. Each DCS implementation is defined by the (set of) parameter signals it is specified for. For designers that want to identify which cores will benefit from DCS, the main difficulty is selecting the set of parameters that results in the most optimal DCS implementation. Assessing whether or not a parameter set is good requires the designer to compare the possible benefits of DCS, a decreased design area and increased design speed, with its overhead. The overhead is introduced by the compilation of the new specialized FPGA circuit and the reconfiguration of the FPGA. However, both the benefits and the overhead are difficult to determine without actually building the DCS implementation. For example, only detailed knowledge of how a RC6 encryption core is described in Register-Transfer Level (RTL) would enable a designer to predict that the area saving of making the key input a parameter is 78.9%. Even this knowledge, by itself, is not enough to determine if DCS would be beneficial; in addition, the parameter set has to change infrequently enough that the overhead is minimal. As a result, the exploration of possible parameter sets takes up a lot of design effort and is typically only done for a (very) small number of parameter sets. This can lead to wrong conclusions on the usefulness of DCS. Cores could be excluded, even though they would have significant gains if a better parameter set was chosen. This could be solved by checking all possible DCS implementations. Since any input signal of the design can be a parameter, the number of possible DCS implementations in a typical application is huge. Even just compiling all possible DCS implementations is infeasible. The challenge then, is how to measure and compare the efficiency of a design and its DCS implementations without actually implementing each of them. In addition, DCS introduces an overhead that has to be taken into account when comparing implementations.

The profiling methodology presented in this article aims to solve these problems. The goal is to help the designer look for the best DCS implementation. It is an automated way to get an overview of the interesting parameter sets, so the designer can spend his effort on those parameters that actually show benefits. The methodology is kept as general as possible, but can easily be adapted to specific use cases. It was also implemented in a tool, the DCS-RTL profiler, that helps the designer tackle this problem in a more automated, structured way. The profiling methodology uses the functional density, a metric to assess a logic circuit first proposed in Wirthlin and Hutchings [1998]. This metric measures the efficiency of an implementation and can be used to compare different implementations of a design. It allows for an honest comparison because the functional density measure incorporates both the gains and overhead of DCS. The problem of the huge number of possible parameter sets is solved by the realization that the majority of the possible DCS implementations can be disregarded by looking only at the dynamic behaviour of their corresponding parameters. This can be done because the overhead of DCS is dependent on the dynamic behaviour of the chosen parameter signal(s).

Our profiling methodology limits the list of good parameter candidates based on the dynamic data collected from the RTL code. This first stage already provides the designer with valuable information on potentially interesting signals. In the next stage, the functional density is calculated for each signal separately. The dynamic behaviour information and the functional density together help the designer to assess the benefits

of a particular parameter choice and allows the designer to quickly see which DCS implementations should be studied in more detail.

Contribution. Our profiling methodology was first presented at the SRCS2012 workshop [Davidson et al. 2012a]. This 4-page article describes the central idea and limited results, based on the first implementation of the DCS-RTL profiler. From now on, this profiler will be referred to as the SRCS profiler. In this article, an extended and significantly improved profiling methodology and profiler are presented and discussed. In addition, the discussion will be extended to include data on more applications, and includes a discussion of the accuracy (Section 4.4). The improvements and additions are also highlighted in Sections 4.1 and 4.2. An overview of the most significant changes is as follows.

(1) The SRCS profiler required an ABC-implemented version of the TMAP tool flow. ABC is a framework for logic synthesis developed by UC Berkley [Berkeley Verification and Synthesis Research Center 2012]. The version presented here is built around an open-source version of the TMAP tool flow available on github [HES Group, Ghent Univeristy 2012]. This new TMAP tool flow is a Java framework for technology mapping that also includes a regular mapping algorithm, MAP. A major improvement of the framework is that it allows an explicit verification that checks if the TMAP mapped circuit still implements the same logic function as the original design. A final benefit is that it will be easier to also open source the RTL-DCS Profiler.
(2) Support for newer FPGA architectures. The SRCS profiler only supported the Virtex-II Pro, which has 4-input Lookup Tables (LUTs). The Virtex-5 FPGAs, and all newer families, use 6-input LUTs. The DCS-RTL profiler presented here supports 6-input LUTs, allowing it to analyse designs targeting newer architectures, such as the Virtex-5 , Virtex-6, and 7-Family devices. The impact of the switch from 4- to 6-input LUTs is mainly in the technology mapping step. Mapping 6-input LUTs results in a significant runtime increase. For example, an 8-bit 32-tap FIR filter is technology mapped in 8.7s on 4-input LUTs and in 95.2s on 6-input LUTs. The newest Altera FPGAs are also dynamically reconfigurable; currently they are not supported, as the TMAP tool flow is not (yet) able to target them.
(3) The parameter pruning was added to improve the execution time. The DCS-RTL profiler now also takes the area overhead of the HWICAP and HWSRL modules into account, while the SRCS profiler did not.
(4) Introduction of ways for the designer to customize the list of parameter candidates. This allows checking only specific parameter candidates, bypassing the automatic list generation.
(5) Significant improvements were made for several of the estimates in the DCS-RTL profiler. In the SRCS profiler, the design clocks were estimated through a critical path analysis of the TMAP result. In this profiler, XST (the Xilinx Synthesis Tool) is used for the clock estimate. This was changed to improve the quality of these estimates and to simplify the support for multiple FPGA families. As the vendor, Xilinx has access to more timing information and can provide a better estimate. In addition, the SRCS profiler estimated the clock only for the parts of design that were implemented using TMAP. The new profiler estimates the clock of the complete design. In other words, the new profiler also includes the unchanged parts of the original design when estimating the clock.
(6) The DCS-RTL profiler was tested on more designs. The extra designs are three Twofish encryption designs (128, 192, and 256 bit), an RC6 encryption and decryption design, and a Pipelined AES core. These designs are all open-source hardware designs, taken from Opencores.org. They were selected based on the availability

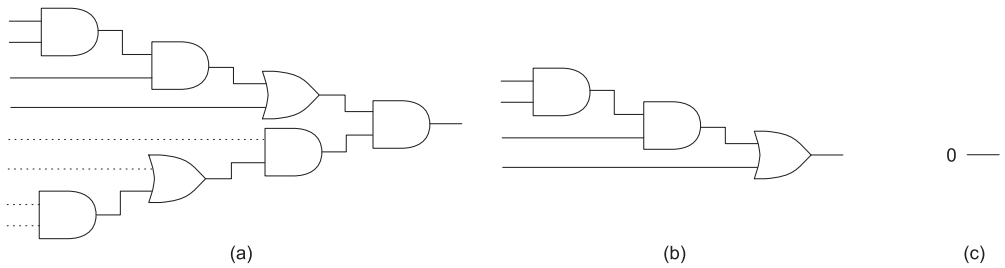


Fig. 1. Partial evaluation and logic removal. All dashed lines are considered constants. In (b) the constants are 1111, and the specialized circuit becomes two AND gates and one OR gate. In (c) the constants are 1001, and the specialized circuit is a constant.

of a test bench. In addition, a regular expression matching design is discussed in Future Work.

The background section (Section 2.1), starts with describing the DCS concept. Next, an overview is given of the TMAP tool flow, a tool flow for DCS developed at Ghent University. Finally, RTL profiling, in general, is discussed. Section 3 details the functional density and the consequences of choosing it as a metric. It also describes the basic profiling methodology we propose. Section 4 presents a profiling tool that implements the methodology discussed in Section 3. The results of three different implementations of this profiler are discussed for 10 designs in Section 4.4, with special attention to the runtime, accuracy, and the quality of the results. To conclude, an overview of our work will be given, including future work to improve the profiling methodology.

2. BACKGROUND

2.1. Dynamic Circuit Specialization

DCC is a technique in which an FPGA circuit is specialized according to the specific values of a set of input signals (the parameters). If a good parameter set is chosen, the specialized circuits can be smaller and/or faster than the original circuit. Each set of parameter values corresponds to a different specialized circuit. If the parameter values change at runtime, the FPGA has to be reconfigured with a new specialized configuration.

The parameters should be infrequently changing signals because each parameter value change will trigger a runtime reconfiguration. Such a runtime reconfiguration incurs a time overhead. If a parameter changes too frequently, the runtime can be increased significantly. Therefore, it is most interesting to choose only the infrequently changing inputs of a design as parameters. Many applications have input signals that change (much) more infrequently than the other inputs. One example thereof is the rule set in Snort, a network intrusion detection system. These rules typically change every few days, at most every few hours, when new vulnerabilities are discovered [Roesch 1999]. For a hardware implementation of Snort, the rules are the best parameter candidates. Another example is an adaptable FIR filter. In many implementations of such a filter, the coefficients change less frequently than the filtered data. In that case, those coefficients are the best parameter candidates.

A straightforward way to generate specialized circuits is to apply constant propagation to the original circuit [Wirthlin and Hutchings 1997]. For each parameter value a specialized circuit can be generated by partially evaluating the original circuit, assuming the parameter value is a constant. Figure 1 illustrates the partial evaluation and the subsequent logic removal. This specialized circuit is smaller than the original circuit but is only valid for one specific parameter value. The decrease in size is highly

design dependent, and for some designs a large part of the logic circuit is removed. For example, in the case of an 8-bit multiplier, assuming one input is constant will allow a 69% LUT-area reduction [Bruneel 2011]. The gains of DCS are not always limited to decreased LUT area; if some of the removed LUTs were part of the critical path, then the specialized circuit will be faster too. Constant propagation is the specialization technique used in this article and is discussed in detail in Section 2.2. Circuit specialization can be extended further than constant propagation. One example is the specialized circuits in Keller [2000]; they are also hand designed so switching between them is optimized.

In a finished implementation, DCS is incorporated as follows: Initially, the FPGA is configured with a circuit specialized for the starting parameter values. The FPGA circuit is processing data until a parameter changes. Then, the new parameter values are used to generate a new specialized circuit or to retrieve one from memory. Next, the FPGA is reconfigured with the new specialized circuit. Finally, the FPGA starts processing data again until the next parameter change, then the specialization process starts over. The specialization process introduces an overhead in the DCS implementation that impacts both the execution time and the required area.

A DCS system can be built by either using precompiled specialized circuits or compiling specialized circuits at runtime. The first solution is only feasible if either the number of parameters or the number of parameter values is small, as it requires storing a specialized circuit for each possible parameter value. The number of possible parameter values is exponential with the total number of parameter bits. Even for a small project, such as a 16-tap adaptable FIR filter with 8-bit coefficients as parameters, generating and storing all specialized circuits (2^{128}) is impossible. The second solution, compiling the circuits at runtime, can be implemented using the traditional RTR tools provided by the FPGA vendors. Lim and Peattie [2002] describe two ways to use these RTR tools. One is "module-based partial reconfiguration," that is, changing the complete functionality of whole regions on the FPGA, and requires the full FPGA tool flow to be run in order to compile one specialized circuit. This compilation takes minutes, and even hours, for complex designs. For instance, in the case of the small adaptable FIR filter already discussed earlier, the compile time was 35s, for a single specialized configuration [Bruneel and Stroobandt 2008]. An overhead of this magnitude is only permissible in very specific cases. The other is "small-bit manipulations," which requires manual intervention with the FPGA editor for each partial bit stream and thus is not feasible as a general approach.

Clearly, precompiling all configurations is infeasible and running the full FPGA tool flow at runtime requires too much time. To solve these problems, the Hardware and Embedded Systems Group at Ghent University has developed the TMAP tool flow [Bruneel 2011]. This tool flow implements DCS efficiently and maximally exploits the existing runtime reconfiguration capabilities of modern FPGAs. It solves all compute-intensive problems at compile time without the need to store large amounts of data. In addition, it allows a circuit to be specialized in hundreds of milliseconds—a much smaller overhead.

2.2. TMAP: A DCS FPGA Flow

The TMAP tool flow is not the only tool flow that uses constant propagation to optimize circuits. Other examples can be found in Bruneel [2011]. It is, however, the first tool flow that automatically generates a specialized circuit from only an annotated Hardware Description Language (HDL) description. The TMAP tool flow implements a symbolic constant propagation. The constants are propagated without any assumptions about the actual constant value, allowing the tool to generate one specialized LUT circuit for all possible parameter values. This LUT circuit is smaller than the generic

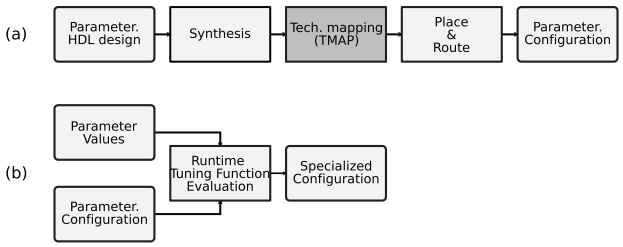
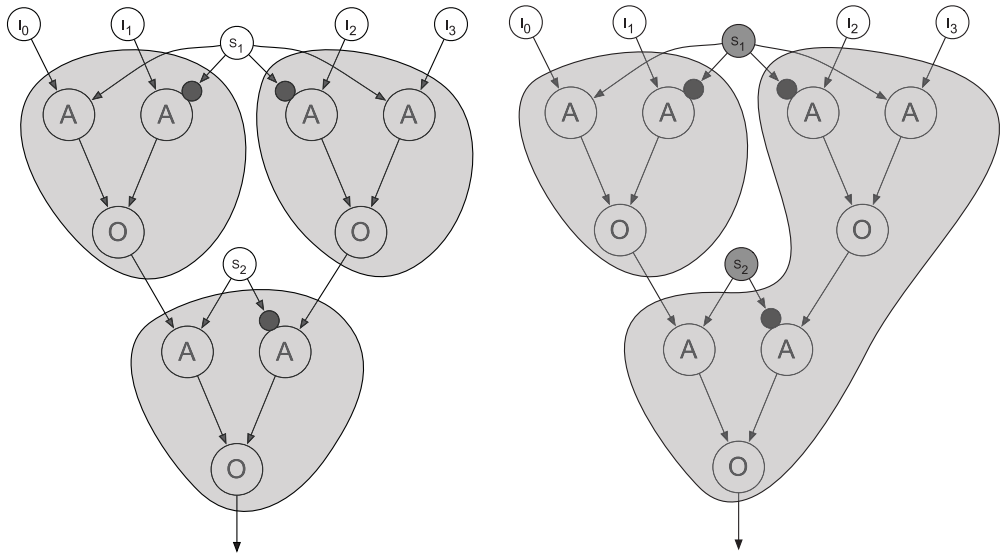


Fig. 2. (a) The TMAP tool flow, based on the standard FPGA flow. The technology mapping was adapted for DCS. (b) The runtime component of the TMAP tool flow.

circuit because the parameters are not considered inputs, but symbolic constants. The functionality that depends on them is now included in the truth tables of some LUTs. These LUTs are called TLUTs and their truth tables change depending on the actual parameter values. In this LUT circuit, the number of LUTs and their connections remain the same for all parameter values. Consequently, the LUT circuit is placed and routed offline, at design time. It is exactly these two steps, placement and routing, that require a lot of time in the regular FPGA tool flow. At runtime, only the TLUT truth tables change, based on the parameter value changes. The specialization procedure calculates the new truth table bits based on the new parameter values, and then the TLUTs are reconfigured.

As also discussed in Bruneel and Stroobandt [2008], TMAP does produce slightly larger specialized circuits than hand-specialized designs or circuits specialized for an actual value. This is because the TMAP constant propagation is symbolic. Since it does not assume an actual value and only LUT truth tables are changing, the structure of the specialized circuit stays the same for all parameter values even if, for the specific parameter values, certain LUTs could be removed completely. In the case of the 16-tap adaptable FIR filter, the coefficients are chosen as the parameters. The size of a circuit specialized for a specific set of coefficient values varies. The largest of these specialized circuits for the FIR filter example requires 1,147 LUTs. The TMAP specialized circuit uses 13% more area (1,301 LUTs). However, the TMAP specialized circuit is 19% faster. In addition, generating new specialized circuits and switching between them requires 35.6s; changing the parameter value of the TMAPped circuit requires only 166μs.

The complete TMAP tool flow is shown in Figure 2. It has similar stages as the traditional FPGA tool flow. It starts from an HDL description of the design, with added annotations (–PARAM) to denote the parameters. The tool flow produces a configuration that contains all static bits of the circuit and a procedure to fill in the LUT values that change based on the parameter values. Compared to a conventional FPGA tool flow, the main difference is a new technology mapping algorithm, called TMAP. This modified mapping algorithm maps a design to a circuit consisting of both LUTs and Tunable LUTs (TLUTs). The result of mapping a 4:1 MUX with a standard mapping algorithm is shown in Figure 3(a). S_0 and S_1 are the selection bits, and $I_{0–3}$ the inputs. The logic function of the MUX is shown in a logic network with AND (A) and OR (O) nodes, and the black dots signify an inverted input. A normal mapping algorithm needs three 3-input LUTs to implement the MUX. In Figure 3(b), the same MUX is shown, only now the selection bits are considered parameters. TMAP not only maps to normal LUTS, but also to TLUTs. TLUTs can have any number of parameter inputs because their truth table bits depend on the parameter value. The relation between the truth table bits and the parameter inputs is expressed as Boolean functions. These Boolean expressions are called the tuning functions. TMAP only requires two 3-input TLUTs to implement the MUX. To place and route the (T)LUT circuit, the TLUTs are treated



(a) Standard Tech. Map. (b) TMAP

Fig. 3. MUX 4:1 technology mapped to 3-input LUTs.

as normal LUTs. After the placement and routing, each tuning function is associated with a specific configuration memory bit. That information, combined with the actual parameter values, is used to compile the reconfiguration bit stream at runtime, as can be seen in Figure 2(b).

In our example, a 16-tap adaptable FIR filter with 8-bit coefficients is 56.6% smaller and 28% faster than the generic FIR filter solution when implemented by TMAP [Bruneel 2011]. TMAP also achieves good results for key-based encryption algorithms. A DCS implementation of TripleDES is 28.7% smaller. Similarly, a DCS implementation of RC6 is 72.7% smaller [Davidson et al. 2011]. The results of the TMAP tool flow are shown in Table III, in Section 4.4.

For our work, we focussed on the TMAP tool flow because it is an efficient and general method for DCS. From now on, unless otherwise stated, DCS refers to DCS as implemented by the TMAP tool flow.

2.3. RTL Profiling

Traditionally, software profilers analyse properties such as memory usage [Ortolani et al. 2011], runtime scaling [Magklis et al. 2003], and the identification of "hot" code [Duesterwald and Bala 2000]. Generally, there are two types of profiling: static and dynamic. Static profiling looks at the code as is, while dynamic profiling is based on data collected from the execution of compiled code. All of the previously mentioned papers, and the profiling methodology presented in this article, implement some form of dynamic profiling.

RTL code profiling is aimed at assessing properties of the hardware the RTL code implements, such as power consumption. Power consumption is partially dependent on how frequently a component (input) switches. Kumar et al. [1995] describes a profiler that takes synthesizable RTL, instruments it, and collects runtime data on how frequently component inputs switch. In Katkoori and Vemuri [1998], a similar profiling methodology is used to find the lowest power RTL description out of a series of RTL

descriptions. The RTL descriptions are generated by High Level Synthesis (HLS) tools; each of the generated RTL descriptions fulfils a set of constraints given to the HLS tools. The profiling is aimed at selecting the one description that uses the least power.

Our profiling methodology also relies on the dynamic behaviour of the compiled design, just like the power-oriented profilers discussed earlier. However, for these profilers only the average number of switches is needed, while our profiling methodology requires additional, more fine-grained, dynamic information, such as the minimal time between transitions. In addition, gathering the dynamic data is only the first stage of the profiling methodology presented here. In a second stage, the functional density that corresponds with each parameter choice is calculated.

3. IDENTIFICATION OF DCS IMPLEMENTATIONS

When identifying designs that benefit from DCS, the main difficulty is the selection of the parameters. The designer has to select a signal, or a group of signals, in the RTL design, which will be parameters. The selection is not straightforward because both the number of parameter candidates is very large and the effects of a particular parameter choice are hard to predict. Before we discuss the parameter selection (Section 3.3), we describe how the efficiency of a DCS implementation can be measured. This section will also discuss the overhead DCS introduces in more detail.

3.1. Functional Density

When the designer is trying to determine which DCS implementation is preferable, a metric for the area-time efficiency is needed that includes both the gains and overhead of DCS. A measure that satisfies these conditions is the functional density; it is the number of computations per unit of area and per unit of time [Dehon 1996]. It was also used to assess the benefits of runtime reconfiguration implementations in Wirthlin and Hutchings [1997].

The functional density captures all effects of the introduction of DCS in the design. However, sometimes a designer only wants to check if DCS improves a single aspect of a design, such as area, execution time, or power consumption. Our profiling methodology is not limited to the functional density; it also allows the designer to check if DCS improved the area or execution time. The DCS-RTL profiler reports on the area that is saved through TMAP and, as the time overhead of DCS is calculated, the DCS execution time is also available. If only the area is important, the best implementation is very clear. When only execution time is considered, the gains of DCS depend on the tradeoff between the time overhead and the speed improvements. The DCS impact on the power depends on the power savings through the reduced design size versus the power needed to runtime reconfigure part of the FPGA fabric. Lorenz et al. [2004] discusses this tradeoff more extensively, but this is not discussed further in our work.

For FPGAs, the functional density is defined by the number of computations (N) performed on an FPGA area A, during the time T. More specifically, for a DCS implementation, this translates to Equation (1), where T_{DCS} is the total execution time of the DCS implementation and A_{DCS} is the total FPGA area used by the DCS implementation.

$$FD_{DCS} = \frac{N}{T_{DCS} \cdot A_{DCS}} \tag{1}$$

It is clear that the implementation with the best area-time efficiency is the one with the highest functional density. To find this implementation more easily, we refine Equation (1) by taking the time overhead of DCS into account. In order to do that, however, we will first discuss the time overhead in more detail.

3.2. DCS Overhead

As discussed earlier, the overall benefits of a DCS implementation depend on the tradeoff between the gains and the overhead DCS introduces. While the gains (reduced design size and improved circuit speed) were already discussed in Sections 2.1 and 2.2, the overhead was not. The overhead is the cost, in both area and time, of the complete specialization process. This process is split up into two subprocesses: the evaluation and the runtime reconfiguration. The former is the process that calculates the new truth table bits, while the latter handles writing to the configuration bits of the FPGA. Both these processes can be implemented in multiple ways; the available choices depend on the targeted FPGA. The following overview assumes a Xilinx Virtex-5 FPGA, but the general concepts apply to all FPGA devices.

3.2.1. Resource Overhead. The specialized circuit is generated by evaluating the tuning functions, which have the parameter values as inputs. An embedded PowerPC, a soft core Microblaze, or a custom core can be used, depending on the preferences of the designer. Some Virtex-5 FPGAs, and most modern FPGAs, have embedded CPUs. If they are present, they are the most interesting choice as they require no extra FPGA area. However, the TMAP tool flow is largely agnostic on how the tuning functions are evaluated. The sizes of and the tradeoffs between the different evaluation options are discussed in detail in Abouelella et al. [2010]. In our profiling tool, the use of an embedded processor is assumed, but it can be easily adapted to a different evaluation platform. If an evaluation platform that requires FPGA LUTs is used, then this is considered part of the resource overhead.

For the runtime reconfiguration, two options exist. The first is the HWICAP module, provided by Xilinx, which gives the designer access to the Internal Configuration Access Port (ICAP) and is available on all Xilinx FPGAs. The ICAP allows an FPGA design to change the FPGA configuration memory frame by frame. The size of the HWICAP module varies significantly for different HWICAP designs. The high-performance XPS-HWICAP uses 714 6-input LUTs and one BRAM on Virtex-5 FPGAs [Xilinx 2011]. In contrast, a low-performance, deprecated, OPB-HWICAP only requires 198 6-input LUTs and one BRAM on the same FPGA family [Xilinx 2006]. In addition, the data width of the ICAP changes for different FPGA families. In a Virtex-II Pro FPGA, it is 8-bit wide and a OPB-HWICAP module uses 224 4-input LUTs and one BRAM [Xilinx 2004]. All later FPGA families have a 32-bit ICAP. In the newest 7-family FPGAs, 544 6-input LUTs and one BRAM are used [Xilinx 2012b]. The ICAP is not very efficient; for example, changing the content of a single LUT truth table requires writing five complete frames to the reconfiguration memory. The limitations of the HWICAP are discussed in more detail in Abouelella et al. [2010]. There is extensive academic research on improving the Xilinx-provided HWICAP modules [Liu et al. 2009; Bhandari et al. 2012; Claus et al. 2010; Manet et al. 2008]. These HWICAP designs exhibit significantly better performance and/or reduced area. Once it is determined that a DCS implementation would be beneficial, any of these could be used to further improve the DCS implementation. In the profiling tool only the original Xilinx, high-performance, HWICAP is taken into account, because we strive to deliver a conservative estimate.

The second option for runtime reconfiguration is to use the shift-register capabilities of LUTs (SRL). When a LUT is used as an SRL, its truth table bits are accessible by the data path and all of its truth table bits form a shift register. An SRL can also still function as a normal LUT. The SRL capability is used to combine all TLUTs in one or multiple shift registers that are only activated during runtime reconfiguration to allow the new truth table bits to be shifted in. The SRL reconfiguration process is much

faster because this method is not restricted by the memory structure of the FPGA. Additionally, the work in Heyse et al. [2012] shows that adding an SRL chain to a design post placement has almost no impact on its performance. While most modern FPGAs have at least some LUTs with SRL functionality, these capabilities vary for different device families. For example, in the Virtex-5 family, only half of the 6-input LUTs can be used as 32-bit SRL memories [Xilinx 2012d]. In the Virtex-II Pro, all LUTs could be used as 16-bit SRLs [Xilinx 2007]. The profiling tool can take into account both Virtex-II Pro and Virtex-5 style SRL architectures. In order to use SRL runtime reconfiguration, a HWSRL module was proposed in Al Farisi [2009]. Its resource overhead is much lower than the HWICAP; it uses 156 4-input LUTs on the Virtex-II Pro. An updated version of the same module requires only 98 Virtex-5 LUTs, but does require 1 BRAM. This is the area overhead the profiling tool takes into account for SRL reconfiguration.

For some applications, extra resource overheads have to be taken into account when implementing them in DCS. For example, if the input data cannot be stopped while the circuit is being specialized and the FPGA reconfigured, then input buffers are needed. These overheads are very application specific; therefore, they are not considered in this work. Their overhead can easily be added to the DCS-RTL profiler if required.

3.2.2. Time Overhead. The time overhead is the total time needed for specialization over the full runtime of the application. It can be expressed as the single specialization time (for specializing the FPGA once) multiplied by the number of times the parameters change.

The *Single Specialization Time* (SST) has two components: the evaluation time and the runtime reconfiguration time. They are both influenced by the number of TLUTs in the DCS implementation. The evaluation is handled by the embedded processor, for the applications considered in this article, targeting a Virtex-5 FPGA—this time is in the order of 10μs. The time needed for the runtime reconfiguration depends on the chosen method for reconfiguration (Section 4.2). The HWICAP is by far the slowest as it includes a lot of overhead bits being sent over the ICAP interface. As a reference, the time needed to reconfigure a XC5VFX70T-1FF1136 Virtex-5 FPGA completely through the HWICAP is in the order of 173ms [Xilinx 2012d]. In DCS, only a (small) part of the FPGA will have to be reconfigured at runtime, never the complete FPGA. The time required to reconfigure a single Virtex-5 frame is 8.1μs.

The *number of times a parameter changes* is determined by the dynamic behaviour of the parameter(s). To take all signals of the parameter set into account, we use the parameter transition profile, which includes any change in the value of any of the signals. The number of parameter transitions determines how many times the SST was incurred.

3.2.3. Refined Functional Density Expression. The functional density expression in Equation (1) can be refined with the time concepts introduced in the previous section. The ratio of T_{DCS} and T_{data}^{total} is given in Equation (2). T_{DCS} is the total execution time of a DCS implementation, including overhead; T_{data}^{total} is the time the FPGA is actually processing data, the execution time without the DCS overhead. On the right-hand side of Equation (2), $\hat{T}_{data}$ is the average time interval for which the parameter value is constant and $\hat{T}_{SST}$ is the average overhead for a single specialization. As the specialization time is the same for each parameter change, this is the SST discussed in Section 3.2.2.

$$\frac{T_{DCS}}{T_{data}^{total}} = \frac{\hat{T}_{SST} + \hat{T}_{data}}{\hat{T}_{data}} \tag{2}$$

The functional density can be further refined by substituting T_{DCS} as in Equation (3).

$$FD_{DCS} = \frac{\hat{T}_{data}}{\hat{T}_{SST} + \hat{T}_{data}} \cdot \frac{N}{T_{data}^{total} \cdot A_{DCS}} \tag{3}$$

The second factor of Equation (3) represents the functional density when the FPGA is never stopped. If DCS introduces an area decrease and a speed improvement, this functional density will be much higher than the functional density of the original design. However, it is an upper bound for the actual functional density of the DCS implementation. A real DCS implementation will always have to be stopped for reconfiguration. The first factor of this equation represents the degradation of the functional density caused by stopping and reconfiguring the FPGA. The degradation is dependent on the ratio of $\hat{T}_{SST}$ (the single specialization overhead) and $\hat{T}_{data}$ (the average time interval between parameter changes). The degradation is small when $\hat{T}_{data}$ is much larger than $\hat{T}_{SST}$, that is, the longer the parameters stay constant, the smaller the degradation. This insight will be used in the next section to aid the selection of good parameter candidates.

3.3. Parameter Selection

In the previous section, the functional density was proposed as the measure for DCS efficiency. To identify whether or not a particular design can benefit from DCS, the set of parameters with the highest functional density has to be found. If this functional density is lower than the functional density of the original design, then DCS cannot improve the design. In that case, the functional density of the DCS implementation might be improved by making changes to the RTL design, which is not in the scope of the work presented here.

The most thorough method for finding the best parameter set is to calculate the functional density for every possible set. Clearly, this is difficult to implement, as the total number of signal sets can be very large for complex designs and scales badly with each additional signal. In order to be feasible, the number of signals has to be reduced. A first reduction is to only consider single signals as parameter candidates; this is a limitation that will be discussed further in Future Work. The number of candidates are then further reduced by analysing the dynamic signal behaviour. As discussed in Equation (3), the degradation of the functional density is influenced by the ratio of the single specialization overhead and the time between parameter changes.

In Figure 4, the functional density of a 16-tap adaptable FIR filter is plotted. The original design uses 2,999 LUTs and has a clock period of 118.4ns. In the DCS implementation, the coefficients were chosen as parameters. The size of the DCS design is 1,315 LUTs and the clock period is 86.8ns. The time needed for one specialization (T_{SST}) is 166μs. The X axis shows the average time between parameter changes ($\hat{T}_{data}$). The plotted curve shows how the functional density is dependent on $\hat{T}_{\text{data}}$. For signals with a longer average time between transitions, the degradation will be lower, resulting in a higher functional density. The functional density of the DCS implementation is already higher than the original design if $\hat{T}_{data}$ is about 1000 clock cycles, or 86 μs. The functional density stops improving once $\hat{T}_{data}$ is 10^5 clock cycles, or 8.86ms. This is about 53 times T_{SST}.

From both Equation (3) and Figure 4, we can conclude that the dynamic property to look for in parameter candidates is the *average time between signal changes*. In our profiler, the parameter candidates are ordered according to this property. In practice, it is very useful to add a filter that removes signals with at least one transition shorter

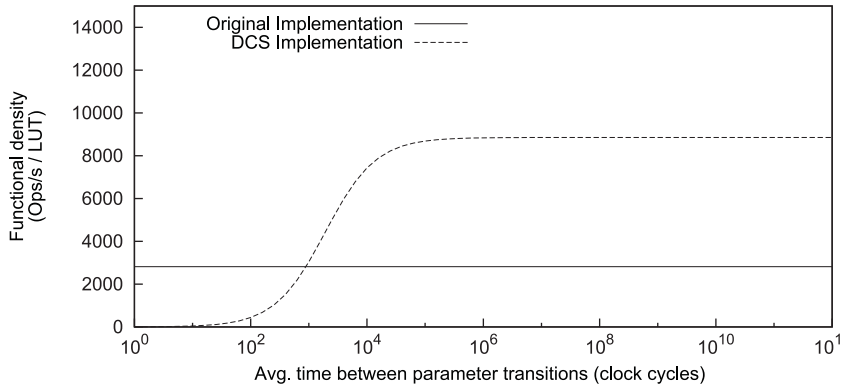


Fig. 4. The functional density of a generic and a DCS implementation.

than a given time interval. The filter allows the removal of signals whose average time between parameter transitions is long enough, but who have a (small) number of transitions that occur too quickly to allow for a reconfiguration phase. An appropriate value for this limit is proposed in Section 4.1.

4. DCS-RTL PROFILER

The DCS-RTL profiler was developed to facilitate the identification of DCS opportunities in RTL designs. The profiler works in two stages. First, it produces a list of parameter candidates based on dynamic signal data collected during simulation (Section 4.1). Next, it provides a functional density overview for each of those parameter candidates (Section 4.2).

4.1. Parameter Candidate Selection and Pruning

In order to identify designs that benefit from DCS, the dynamic behaviour of both the inputs and the internal signals has to be analysed. The dynamic behaviour is needed both to find good parameter candidates and to be able to calculate the functional density of each candidate. The signal behaviour is not only a function of the RTL description, but also of the dynamic behaviour of the simulation input. It is hard to get an overview of the dynamic behaviour of the complete design, across all levels, which is needed to allow a good parameter selection. Therefore, the RTL-DCS profiler analyses the dynamic signal behaviour of a complete design based on a ".vcd" file. This file contains a log of the internal design signals and can be generated by most RTL simulators. RTL simulators are typically used to verify the correctness of the RTL implementation throughout the design process. The ".vcd" file is used because the standard simulator interface does not allow an easy identification of signals with a high average time between transitions, nor does it give a good overview of all signals in a design. If a test bench is provided, the profiler generates its own ".vcd" file using the ModelSim [Graphics 2008] RTL simulator.

The analysis in Section 3.3 shows clearly that the number of candidates needs to be reduced and that the average time between signal transitions is the property that should be used to rank the parameter candidates. A first reduction is to only consider single signals as parameter candidates. Sets of signals are not considered; this limitation is planned to be resolved in future work. In addition, all parameter candidates are filtered to eliminate signals that are sure to result in a high functional density degradation. Any signal that has two transitions closer together than a predefined limit is excluded from the analysis. A good choice for the limit is the SST for one TLUT

because then signals with a high functional density degradation are excluded even when they only introduce the smallest possible overhead. The time needed to reconfigure one TLUT is about 40.94μs for the HWICAP. In practice, the exclusion of parameter sets and the introduction of the limit reduces the number of parameter candidates to a manageable amount.

Designers can customize the list of parameter candidates in two ways. Signals can be included and excluded from the analysis through annotation of the RTL description.

Excluding signals is done through a "–NOPARAM" annotation and is useful to remove reset, start, and ready signals from the parameter list. These signals change infrequently, but should not be made parameters, as a transition on these signals generally should be processed instantly.

Similarly, adding a "–PARAM" annotation to a signal in the RTL design will force the RTL-DCS profiler to add it to the list of parameter candidates. However, it is also possible to bypass generating the list completely. In that case, the annotated signal(s) are the only candidates. This allows the designer to explore "what-if" scenarios. If the designer already suspects certain signals would be good parameters, the profiler can be used to verify if they actually are.

4.1.1. Parameter Pruning. The parameter selection already limits the list of parameter candidates for which the functional density is calculated. In addition, a pruning of the parameter candidates can be applied during the functional density calculation. For candidates that are sure not to lead to a functional density increase, the functional density calculation is aborted. Whether or not to prune a candidate is determined after the technology mapping of the DCS implementation with TMAP. Candidates that cannot lead to a functional density increase either do not decrease area enough to offset the DCS area overhead (i) or do not decrease the longest path (ii).

Parameter candidates that do fulfil either (i) or (ii) could result in functional density increases, so for these signals the functional density calculation is continued. The signals that only fulfil (ii) have to be checked as they could lead to a shorter critical path of the full design. The area overhead used in condition (i) is the minimal area overhead of DCS, the area of the HWSRL module.

4.2. Functional Density (FD)

The second stage in the DCS-RTL profiler is calculating the functional density for each parameter candidate. First, the functional density of the original design is calculated, using Equation (1). The area of the design, A_{orig}, can easily be determined by mapping the design using a conventional mapping algorithm, MAP, which is part of the TMAP flow framework. If the design produces an output value each clock cycle, then the number of computations (N) is equal to the number of clock cycles and T_{orig} (the execution time of the original design) can be expressed as $N \cdot t_{orig}$, with t_{orig} the clock period of the design. If outputs are produced at a slower rate, $T_{orig} = R \cdot N \cdot t_{orig}$., with R the average number of clock cycles it takes before the design produces a result. The functional density of the original design then becomes Equation (4). The clock of the design is collected by running the design through the Xilinx FPGA tool flow [Xilinx 2012a].

$$FD_{orig} = \frac{1}{t_{orig} \cdot A_{orig} \cdot R} \tag{4}$$

The functional density of a DCS implementation of the design is calculated using Equation (3). A_{DCS} is determined by mapping the design with TMAP. The area overhead for HWICAP or HWSRL reconfiguration is added to A_{DCS}. Comparing A_{orig} and A_{DCS}

shows the area saving DCS introduced. Equation (5) is Equation (3) with the following substitutions:

—T_{data}^{total}, rewritten as $R \cdot N \cdot t_{DCS}$, with t_{DCS} the clock period of the DCS implementation, and N and R as in the substitution for Equation (4).

—$\hat{T}_{data}$ is the time between parameter changes in the DCS implementation. This time is not exactly the same as the average time between parameter changes in the original implementation ($\hat{T}_{data}^{orig}$) but can be expressed as a function of it using the average number of computations between parameter changes and the clock periods of both implementations (t_{DCS}, t_{orig}). $\hat{T}_{data}^{orig}$ is the dynamic data collected for each signal in the previous stage of the RTL-DCS profiler.

$$FD_{DCS} = \frac{\hat{T}_{data}^{orig} \cdot \frac{t_{DCS}}{t_{orig}}}{\hat{T}_{SST} + \hat{T}_{data}^{orig} \cdot \frac{t_{DCS}}{t_{orig}}} \cdot \frac{1}{t_{DCS} \cdot A_{DCS} \cdot R} \tag{5}$$

The only unknown variable in Equation (5) is $\hat{T}_{SST}$, which is discussed in detail in the following, as determining it is more complex.

4.3. Single Specialization Time (T_{SST})

This section describes the implementation of the time overhead discussed in Section 3.2.2. The specialization consists of two processes: evaluation and reconfiguration. The *evaluation overhead* can be determined by analysing the tuning functions, which express how each TLUT bit is dependent on the parameter value. The number of Boolean operations ($BoolOps$) in the evaluation process is counted and multiplied by the average time required for a single Boolean operation on the evaluation platform (T_{BoolOp}).

$$T_{evaluation} = BoolOps \cdot T_{BoolOp}. \tag{6}$$

The tuning functions are currently expressed using only AND and NOT operations. Both operations require the same time on a PowerPC 440, embedded in the XC5VFX70T-1FF1136 Virtex-5 FPGA. For this embedded processor, T_{BoolOp} is 1.04 clock cycles.

The *reconfiguration time* greatly depends on the chosen reconfiguration method: HWICAP or HWSRL. In the case of the HWICAP reconfiguration, the FPGA is reconfigured frame by frame. For the Virtex-5 family, a single frame is reconfigured in 8.1μs (Equation (7)).

$$T_{reconfiguration}^{HWICAP} = \#frames \cdot T_{frame} \tag{7}$$

The number of frames that has to be reconfigured is dependent on the exact location of each TLUT in the FPGA fabric. These locations are assigned during placement, which is done by the Xilinx FPGA tool flow [Xilinx 2012a].

In SRL reconfiguration, all the TLUTs are combined in one register chain, which is only active during reconfiguration. This method of reconfiguration is much faster because only the actual truth table bits are sent. The SRL chain is clocked at the design speed, which is only known exactly after routing. The *reconfiguration time* using SRLs can be calculated easily by combining the number of TLUTS, the truth table size (2^6 for the Virtex-5 Family), and the clock speed of the DCS implementation (Equation (8)).

$$T_{reconfiguration}^{SRL} = \#TLUTS \cdot 64 \cdot t_{DCS} \tag{8}$$

Using Equations (5), (6), and (7) or (8), the functional density of the original design and the DCS implementation can be calculated. However, to have all the information

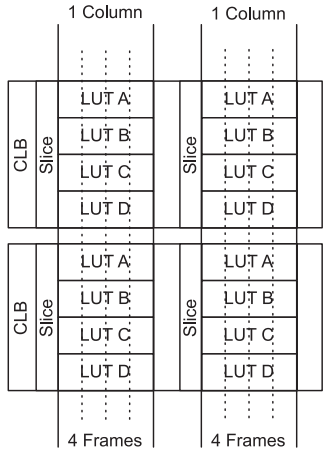


Fig. 5. The Virtex-5 configuration memory structure.

necessary for this calculation, the complete Xilinx FPGA tool flow has to be run. Placement and routing are the most time-consuming steps in the Xilinx tool flow [Ababei 2009] and are necessary to determine the exact value of t_{DCS}, t_{orig} in Equation (5) and $\#frames$ in the HWICAP T_{SST} calculation. If we want to improve the runtime of the profiler, then avoiding placement and routing will have the most significant impact. However, this means t_{DCS}, t_{orig} and $\#frames$ will have to be estimated.

Estimating t_{DCS} and t_{orig}. The clock period of the DCS implementation (t_{DCS}) and the clock of the original implementation (t_{orig}) is estimated using XST [Xilinx 2012c], the synthesis tool that is part of the Xilinx FPGA tool flow. XST synthesizes the result of both the original design and the DCS implementation for the target FPGA and estimates a clock speed in each case.

Estimating #frames. As the placement and routing steps are avoided, the exact locations of the TLUTs are unknown. Therefore, the number of FPGA frames that contain TLUTs has to be estimated. To be able to estimate this number, we have to take a closer look at the Virtex-5 FPGA configuration memory architecture (Figure 5).

In the Virtex-5 FPGA family, one LUT truth table is spread over four frames[1] [Xilinx 2012d]. There are four LUTs in one slice and 20 slices are grouped together in one tile. A tile is one slice wide. So, each tile has 80 LUTs whose truth tables are scattered over the same four frames.

As soon as one LUT in a tile is a TLUT, all four corresponding frames have to be reconfigured and sent to the FPGA. The frame estimate then reduces to estimating how many tiles will contain at least one TLUT. For the estimate, it is assumed that the complete design is implemented in a perfectly square area of FPGA fabric; this determines the maximum number of tiles in the design as $C = \lceil\sqrt{LUTs}\rceil \cdot \lceil\frac{\sqrt{LUTs}}{80}\rceil$. A tile is also considered to be completely used by the design as soon as one of its LUTs would be part of the design. A third assumption is that the TLUTs are spread out uniformly over all LUTs in the tiles. Under these assumptions, the number of tiles containing at least one TLUT is given by Equation (11), with C the total number of

[1]This means reconfiguring one TLUT requires the writing of five frames, the four truth table frames, and a padding frame. The time to reconfigure these frames is the limit chosen to filter parameter candidates in Section 4.1.

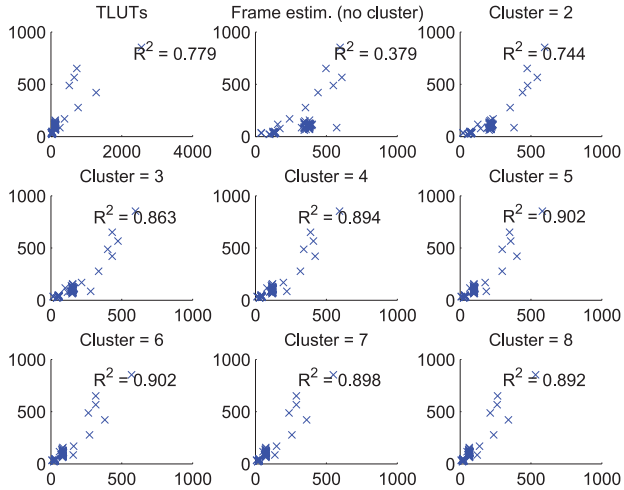


Fig. 6. R^2 values for each clustering option, and the number of columns.

tiles and *tlut* the total number of TLUTs.

$$E[\#Tiles] = C \cdot P(tlut \in Tile) \tag{9}$$

$$= C \cdot (1 - P(tlut \notin Tile)) \tag{10}$$

$$= C \cdot \left(1 - \prod_{n=0}^{tlut-1} \left(\frac{(C \cdot 80 - n) - 80}{C \cdot 80 - n}\right)\right) \tag{11}$$

The probability of a tile not containing a TLUT ($P(tlut \notin Tile)$) is the probability none of the TLUTs are part of that tile. This probability is the product of the probability of each TLUT to be in that tile. A tile contains 80 LUTs of the design and, under the uniformity assumption, all LUTs in the design are equally likely to be a TLUT. The probability of the first TLUT being in the column is the total number of LUTs in the design not part of the tile ($C \cdot 80 - 80$) divided by the total number of LUTs ($C \cdot 80$). For the second TLUT there is one less LUT available and this probability becomes $\frac{C \cdot 80 - 1 - 80}{C \cdot 80 - 1}$.

After experiments, both assumptions at the basis of Equation (11) have to be modified. The predicted number of frames is too large, especially for small numbers of TLUTs, because they are treated as completely unrelated. This solution is shown in the middle scatter plot in the top row of Figure 6. In most designs, however, TLUTs occur together and are interconnected. If a tile contains one TLUT, it is more likely another TLUT will also be part of it. This was taken into account by clustering the TLUTs; each cluster of TLUTs is placed into a tile as a group. Figure 6 shows the correlation factors of several possible cluster sizes with the actual number of frames of all parameter candidates in all designs. The first correlation is between the number of TLUTs and the number of frames ($R^2 = 0.76$). The other correlation factors are of the estimate (Equation (11)) with different cluster sizes. A cluster size of five results in the highest correlation over all designs ($R^2 = 0.902$). In cases where the number of TLUTs is a large fraction of the total number of TLUTs, the number of frames that contain TLUTs is underestimated because of the assumption that the design is placed in a perfectly square region. All designs, except for the RC6 designs, are placed in an area that has a higher width than under the squareness assumption. A similar

analysis as for the TLUT clustering has shown that the best estimates are obtained if the design is considered to be 1.7 times wider. This correction was done for the data in Figure 6; $R^2 = 0.902$ is the highest correlation achieved over all cluster sizes and column corrections.

The frame estimation is quite difficult; the estimate is made based on only the number of LUTs and TLUTs in a design. In addition, there is a lot of variability between designs and within designs, even for similar parameters. For example, in the 8-bit 16-tap FIR filter a typical coefficient results in a design of 2,039 LUTs, 41 of which are TLUTs. The corresponding number of frames that contain TLUTs after placement and routing varies between 29 and 41, a difference of 40%. Trying to improve the correlation for a certain design type, such as the FIR filters, deteriorates the correlation for the other designs.

The functional density can be calculated using Equation (5). To do this, data is collected from either the complete FPGA tool flow or XST, depending on whether the clocks and the number of frames are estimated or calculated. A brief overview of the tools that are used is as follows:

MAP: maps the original design, delivering A_{orig}. It is a Java implementation of a standard mapping algorithm [Hauck and Dehon 2007].

TMAP: maps the original design, with the parameter candidate as the parameter. It provides A_{DCS} and information necessary for computing T_{SST} ($BoolOps$, #TLUTS). TMAP does not have to be applied to the complete design. If the design is structured in different entities, TMAP can act on the level of these separate entities. The tool flow reinserts an RTL description of the TMAP mapped entity into the original top level of the design. This TMAPped entity is annotated and will not be altered by XST, the Xilinx technology mapper [Xilinx 2012c]. The altered RTL description is passed to the Xilinx FPGA tool flow as the DCS implementation. More practical details on the TMAP tool flow can be found in the user guide on github [HES Group, Ghent Univeristy 2012].

The Xilinx FPGA Flow: compiles both the original design and the DCS implementation. After compilation t_{DCS}, t_{orig} and the exact number of frames to be reconfigured ($\#frames$) can be extracted. For the following experiments, version 13.4 of the Xilinx FPGA flow was used. When estimating the functional density, only the synthesis part of this tool flow (XST) is used.

4.4. Results

In order to assess the runtime, the accuracy, and the quality of the profiler results, three profiler implementations were run on 10 designs in total. In all of these cases, the parameter selection happened automatically, with the limit discussed in Section 4.1. All versions of the RTL-DCS profiler were run on a computer with 32 GBs of RAM and an Intel Core i7-3770 3.40GHz CPU. A Virtex-5 FPGA (XC5VFX70T-1FF1136) was chosen as the target FPGA, but any other Xilinx FPGA could have been chosen. First, an overview of the designs is given; next, the execution time and the accuracy of the estimates is discussed in detail; and finally, the quality of the results is also analysed.

FIR16, 32. These are four adaptable FIR filter designs. Two with 8-bit coefficients (16 and 32 taps) and two with 16-bit coefficients (16 and 32 taps). The input is always of an equal size as the coefficients. They were designed by the Hardware Embedded Systems group at the Ghent University.

Twofish 128, 192, 256. Three designs that implement the Twofish encryption algorithm. Each encrypts 128 bit of data using a fixed key size. There are three possible key sizes (128, 192, and 256 bits) [opencores 2012c]. This implementation of the twofish algorithm is round based and sequential, and it reuses the same hardware to calculate all 16 consecutive encryption steps. An encrypted version of the input is generated every 16 clock ticks. This is a nonpipelined implementation.

Table I. Runtimes of All Three RTL-DCS Profiler Implementations

Design	Orig. size (LUTs)	# cand.	After prun.	Total runtime (h:m:s) Exact FD	Exact FD (prun.)	Est. FD	Runtime Impr.
16-tap FIR filter (8-bit)	2099	17	1	0:45:47	0:14:31	0:12:31	3.65×
32-tap FIR filter (8-bit)	4399	33	1	2:38:19	1:04:09	1:02:33	2.54×
16-tap FIR filter (16-bit)	8977	17	17	2:38:35	2:38:35	1:39:39	1.59×
32-tap FIR filter (16-bit)	17312	33	32	10:34:20	10:20:45	7:19:50	1.41×
RC6 encryption	2772	2	1	0:18:25	0:16:26	0:13:47	1.33×
RC6 decryption	3017	2	1	0:19:26	0:17:21	0:14:42	1.32×
Twofish 128	5491	52	14	4:38:10	1:27:13	0:33:09	8.3×
Twofish 192	6891	69	19	7:05:35	1:56:29	0:47:34	8.9×
Twofish 256	8270	86	23	9:40:33	2:29:08	1:01:24	9.4×
Pipelined AES	12958	15	5	2:18:58	0:57:20	0:29:23	5.19×

RC6 encryption/decryption. Two designs: the first implements RC6 encryption, and the other core is an implementation of RC6 decryption [opencores 2012b]. Both cores read in 64 bits of data in four clock cycles which are encrypted or decrypted using a 16-bit key. These cores are not pipelined and there is no hardware reuse.

Pipelined AES. A pipelined, round-based, implementation of the AES algorithm with a 128-bit key [opencores 2012a]. The design takes 30 clock ticks to generate an encrypted version of 128 bits of data. Once the pipeline is filled, 128 bits of encrypted data is generated every clock tick.

4.4.1. Profiler Runtimes. The runtimes of the different RTL-DCS profiler implementations are shown in Table I. The standard implementation ("Exact FD") works as described in the previous section, except for the pruning, which is not done in this profiler. The runtime for the unoptimized profiler ranges from just under 20 minutes for the fastest design to over 10.5 hours for the largest design. By far, the largest fraction of this time is the time spent in the FD calculation for each parameter candidate. Finding the parameter candidates has a negligible impact on the total execution time of the profiler. For example, in the 8-bit 16-tap FIR filter design, the first step only counts for 1.13% of the total execution time and this holds for larger designs as well. Therefore, our focus in this section is not on reducing the runtime of the parameter candidate extraction, but on reducing the time required for the functional density calculation. The main part of this time is spent to run the complete Xilinx FPGA flow for each parameter candidate.

We present three profiler implementations. The first one, under "Exact FD" is already slightly optimized, by making sure data can be reused by FD calculations of different parameter candidates. For example, if several signals of the same entity are candidates, it will only be mapped by MAP once, and the result is reused in later FD calculations. The complete Xilinx FPGA tool flow is run for each parameter candidate.

In the second implementation ("Exact FD (prun.)"), our pruning method, as described in Section 4.1.1, is used. The functional density calculation is stopped when it is clear the parameter candidate will not introduce a functional density gain. This removes the need to run the Xilinx tool flow for the pruned parameter candidate. Thus, the most time-consuming steps of the functional density calculation are only done for the candidates that could show an actual functional density improvement.

The third implementation, under "Est. FD" in Table I, includes our pruning method and the functional density estimate. Here, the most time-consuming steps of the Xilinx

Table II. Accuracy of the FD Gain Estimates

Design	Clock period error (% difference)			Avg. Abs. FD gain error			
	Orig. design	Avg. DCS		HWICAP		SRL	
		Min.	Max.	Min.	Max.	Min.	Max.
16-tap FIR filter (8-bit)	−30.3%	−42.3%		5.5%		6.9%	
32-tap FIR filter (8-bit)	−39.7%	−66.0%		11.7%		13.1%	
16-tap FIR filter (16-bit)	−43.6%	−1.8%	−37.0%	0.7%	−7.5%	−6.2%	−8.0%
32-tap FIR filter (16-bit)	−45.2%	0.3%	−31.4%	−5.0%	13.4%	−5.2%	−11.3%
RC6 encryption	−6.4%	8.1%		−7.7%		−27.2%	
RC6 decryption	−0.9%	11.9%		−18.7%		−26.7%	
Twofish 128	−7.4%	−2.1%	4.6%	0%	2.2%	0%	1.7%
Twofish 192	−3.1%	−1.8%	−10.9%	0%	2.1%	0%	1.1%
Twofish 256	−1.2%	−1.9%	−7.4%	0%	1.5%	0%	1.2%
Pipelined AES	−62.8%	−59.5%		7.1%	13.3%	0.8%	23.4%

FPGA tool flow are completely avoided for all parameter candidates. The functional density is estimated using the XST clock period estimates and the frame estimate.

As Table I shows, the unoptimized profiler takes a long time. In most cases, the runtimes are already greatly reduced by the introduction of our pruning step. The designs with the most gain from pruning are those that have a large number of candidates with small area decreases (both 8-bit FIR filters), because those candidates will be discarded in the pruning. For those designs, the functional density estimation decreases the runtime only slightly. Designs that still have a larger number of candidates after pruning benefit the most from avoiding the placement and routing (e.g., all Twofish designs).

Reviewing the data for all the designs, we conclude that the size of the design, in itself, has no clear impact on the relative decrease in execution time. Rather, the design size in combination with the number of parameters is a better indicator of the runtime. This is not always the case, as the execution time is also dependent on what part of design the parameter(s) are located in. A large design with a lot of candidates in small subentities can be analysed faster than a smaller design that has one or two top-level input candidates. In addition, the type of design also has a large influence on the overall runtime.

While some larger designs have a smaller relative decrease of the runtime, this still results in large absolute runtime decreases. For the most time-consuming design, the 16-bit 32-tap FIR filter, the exact FD calculation takes 10.5 hours, while the estimated FD calculation takes 7.4 hours, reducing the runtime by about a third. Except for the 16-bit FIR filters, the profiler with the estimated FD succeeds in analysing all designs in 1 hour or less.

The decreases in execution time discussed earlier come at a cost: the FD estimation makes the profiler less accurate. In the next section the accuracy of the estimate and its impact on the profiler results will be discussed.

4.4.2. Accuracy. The estimated values in the profiler are the clock periods of both the original and the DCS implementation (t_{DCS} and t_{orig}) and the number of frames that have to be reconfigured in HWICAP reconfiguration. The number of frames estimate was discussed more extensively in the previous section. Here, the clock period estimate will be discussed first, then the resulting final FD estimate will be discussed in more detail.

The *clock period* estimates (t_{DCS} and t_{orig}) are used for calculating the FD for both HWICAP and SRL reconfiguration. The exact clock period is determined by running the complete Xilinx FPGA tool flow, while for the clock period estimate only XST, the Xilinx synthesis tool, is run. The results of both tools are compared in Table II. It

Table III. FD Gain for the Best Parameter Candidate of the FIR Filter and RC6 Designs

Design	Orig. design LUTs	TMAP (no overhead) LUTs	Exact FD gain HWICAP	Exact FD gain SRL	Estimated FD gain HWICAP	Estimated FD gain SRL
16-tap FIR filter (8-bit)	2099	1235	8%	72.2%	−5.8%	47.8%
32-tap FIR filter (8-bit)	4399	2671	−18.2%	10.1%	−0.4%	38.6%
16-tap FIR filter (16-bit)	8976	4176	18.8%	66.1%	28.3%	67.4%
32-tap FIR filter (16-bit)	17190	7591	4.5%	36.9%	13.3%	42.5%
RC6 encryption	2772	585	51.5%	167.4%	36.3%	132.8%
RC6 decryption	3017	656	14%	79.5%	43.5%	130.5%
Twofish 128	5490	4265	−98,7%	−84,3%	−96,5%	−82,6%
Twofish 192	6889	4490	−98,4%	−84,1%	−96,3%	−83,0%
Twofish 256	8269	6229	−98,8%	−90,3%	−97,3%	−89,1%
Pipelined AES	12954	9082	−88,5%	−66,4%	−81,4%	−63,5%

lists both the error for the original designs and the error for the DCS implementations of each design. If there is more than one parameter candidate left after pruning, the minimum and maximum error is shown. A negative value is an underestimate of the clock period, resulting in an overestimate of the actual clock speed. For the original designs, the accuracy of the clock estimates varies greatly. It is accurate for the RC6 decryption design (−0.9%), but inaccurate for the FIR filter designs (−30% to −45.2%), with a very bad estimate for the AES design (−62%). Generally, the clock period of the original designs is underestimated by XST. For the DCS implementations, XST has similar results. The spread of the minimum and maximum error is design dependent. For the Twofish designs, this spread varies between 9.1% and 5.5%; for FIR filters it is much wider (35.8%). This could be because the parameter candidate in the FIR designs results in a much larger area reduction. When the results over several DCS implementations are averaged, the result is better (Twofish and Pipelined AES). To conclude, the clock estimates produced by XST are not very reliable. However, in most designs similar errors are made for the original and the DCS design, reducing the impact of these inaccuracies on the functional density gain estimate.

The goal of the profiler is to deliver an accurate estimate of the *functional density* gained through the introduction of DCS. To do this, all profiler implementations calculate the relative functional density increase of the DCS implementation compared to the original functional density. This is done for both HWICAP and SRL reconfiguration. The last four columns of Table II show the accuracy of the estimate-based profiler. The functional density gain is a percentage value; these columns show the absolute difference between the estimated and the exact functional density gains. An estimated functional density gain of 35% that is actually 30% when calculated, would result in a 5% value in these columns. A negative value is an underestimate of the actual functional density gain. The HWICAP estimate is influenced by both the clock and the frame estimates, and the HWSRL estimate only by the clock estimate. If multiple parameter candidates remain after pruning and they have different errors, then the minimal and maximum errors are shown. The estimate for the Twofish designs is so accurate because the functional density decreases significantly when DCS is applied to this design. The decrease is caused by a low area savings in combination with a parameter candidate that changes too frequently, which results in the clock and frame estimates having very little influence on the overall functional density estimate.

The largest errors are for the RC6 encryption designs, for which the predicted gains are lower than the actual gains. This is mainly caused by XST underestimating the clock period of the original design, and overestimating the clock period of the DCS implementation. In the HWICAP estimate this is partially offset by the fact that the

actual number of frames to be reconfigured is underestimated, which is not the case for the HWSRL estimate.

4.4.3. Quality. While in the previous section the accuracy of the estimates was discussed, here we will focus on the parameter selection aspect. As the previous section shows, the estimation can differ significantly from the actual functional density. However, the actual parameter selection does not differ significantly for both the estimated and the exact profiler implementations. The designs fall into two broad categories: (i) designs that show good properties for DCS (FIR filters and RC6) and (ii) designs that are clearly not suited as DCS implementations (Twofish and Pipelined AES). Both implementations place all designs in the same categories. In addition, for each design in category (i), both implementations arrive at the same parameter selection. The estimate-based profiler even succeeds in detecting when the HWICAP shows no gains from DCS and HWSRL implementation does. Each design is discussed separately as follows.

FIR Filters. In all four FIR filters, the coefficients are on the list of parameter candidates. They are pruned in the 8-bit FIR filters, but even in the 16-bit FIR filters a single coefficient never reduces area more than 3.9%. All the adaptable FIR filter designs show one other parameter candidate: the input to the multiplier instance in each of the taps. This parameter candidate leads to very significant area decreases in the two 8-bit (41.2%, 39.3%) and the two 16-bit FIR filters (53.5%, 55.9%). The only FIR filter that does not show a functional density gain with HWICAP reconfiguration is the 32-tap 8-bit FIR filter (−18.2%). In contrast, SRL reconfiguration does show a clear gain: 72.2% and 10.1% for both 8-bit FIR filters, and 66.1% and 36.9% for the 16-bit FIR filters. The gains for the 32-tap FIR filters are lower, as the number of TLUTs increases linearly with the number of taps (24 LUTs per tap), increasing the single specialization time, while the relative area reduction is similar. The same test data and test bench were used for all FIR-filter designs.

RC6 Encryption and Decryption. In both the encryption and decryption designs only two signals have the required dynamic behaviour to be considered parameter candidates. One is an input of a carry-lookahead adder, saves very little area (0.07%), and is pruned in our more optimized profiler implementations. The other is the key input, which leads to very significant area decreases: 78.9% in the encryption design and 78.3% in the decryption. These area decreases translate to an actual functional density increase for HWICAP reconfiguration in RC6 encryption (51.5%) and decryption (14%). The gain for the decryption design is lower, because it contains more TLUTs than the encryption design, resulting in a larger specialization time overhead.

Pipelined AES. In this design, only five parameter candidates remain after pruning. They show an area decrease from 9 to 29.9%, but do not lead to an increased functional density. From studying the design, it is clear all remaining parameters are key-input related. The key is also one of the candidates, but it is pruned because of its low LUT area decrease. The low area decreases are caused by the pipelining of the key expansion. The TMAP mapper does not consider the output of a FF with a parameter at the input to be a parameter. This limits the area savings it is able to obtain. However, work is being done to improve this. This can also be solved by rewriting part of the design, so the key is not pipelined any more.

Twofish. For these three designs, the profiler shows that only a small number of signals have the required dynamic signal behaviour to be considered parameter candidates. This fits with the way the Twofish algorithm is implemented: this is a sequential, round-based design. The same hardware is reused for multiple clocks to calculate one

output value. This is an area savings technique that trades performance for area. It also causes large parts of the designs to be dependent on signals that change almost every clock cycle, rendering these designs unsuitable for DCS implementation.

The RTL-DCS profiler succeeds in identifying opportunities for DCS, with two additional comments. First, the FD gain estimate is not only dependent on the RTL description of the design, but also on the dynamic signal behaviour. This behaviour is extracted from a test bench, which means that this test bench has a big impact on the parameter identification. To get a good result, the test bench should be a good reflection of the actual use case of the design. Consequently, a test bench only written for verification purposes is generally not a good choice. Secondly, the profiler analyses a particular RTL design, not an application. For example, implementing AES differently can lead to much improved DCS gains [Davidson et al. 2011].

5. CONCLUSIONS AND FUTURE WORK

This article presents a profiling methodology that identifies opportunities for DCS, a task that is difficult for the designer. In addition, results on three profilers that implement this methodology are discussed in detail. All three profiler implementations were validated by analysing 10 designs. The results of these tests show that the functional density gain estimation does not preclude the identification of good parameter candidates. Both improved profiler implementations reduce the execution time significantly: the estimate-based profiler achieves a reduction of over 8× for three designs. All designs, except for the three largest FIR filters, are analysed in less than 1 hour.

In future work, the clustering of parameter candidates is an interesting addition. The current profiler considers each candidate on its own. However, making two candidates a parameter at the same time could increase both the area decease and the overhead. The tradeoff between these two should be researched more extensively. An example design that shows this clearly is a regular expression matching block, described in more detail in Davidson et al. [2012b]. This block matches regular expression syntax on one character and can be chained together to implement regular expressions that contain character groupings. The profiler does not succeed in finding a gain from DCS in this case. All candidates introduce either no area savings or an area reduction of one or two LUTs out of 40. Our earlier experience informs us that there is a good parameter selection for this design; however, it requires several parameters to be considered together. All signals of this good selection are found in the list of parameter candidates. However, selecting the good set is not easily possible from the output of the profiler; just combining all candidates with an area decrease will not necessarily lead to the set with the best DCS gain.

REFERENCES

Cristinel Ababei. 2009. Speeding up FPGA placement via partitioning and multithreading. *Int. J. Reconfig. Comput.* 2009, Article 6.

Fatma Abouelella, Karel Bruneel, and Dirk Stroobandt. 2010. Towards a more efficient run-time FPGA configuration generation. In *Proceedings of the International Conference on Parallel Computing (ParCo'09)*. Lyon, France, 8.

Brahim Al Farisi. 2009. *Herconfiguratie van LUT's via hun schuifregisterfunctionaliteit*. Master's thesis. Universiteit Gent.

Berkeley Verification and Synthesis Research Center. 2012. ABC: A System for Sequential Synthesis and Verification.

Sheetal Bhandari, Fabio Cancare, Davide Basilio Bartolini, Matteo Carminati, Marco Domenico Santambrogio, and Donatella Sciuto. 2012. On the management of dynamic partial reconfiguration to speed-up intrinsic evolvable hardware systems. In *Proceedings of the 6th HiPEAC Workshop on Reconfigurable Computing*.

Karel Bruneel. 2011. *Efficient Circuit Specialization for Dynamic Reconfiguration of FPGAs*. Ph.D. dissertation. Ghent University.

Karel Bruneel and Dirk Stroobandt. 2008. Automatic generation of run-time parameterizable configurations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, U. Kebschull, M. Platzner, and Teich J. (Eds.). Kirchhoff Institute for Physics, Heidelberg, 361–366.

Christopher Claus, Rehan Ahmed, Florian Altenried, and Walter Stechele. 2010. Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 55–67.

Tom Davidson, Fatma Abouelella, Karel Bruneel, and Dirk Stroobandt. 2011. Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment. *Int. J. Reconfig. Comput.*

Tom Davidson, Karel Bruneel, and Dirk Stroobandt. 2012a. Identifying opportunities for dynamic circuit specialization. In *Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS'12)*.

Tom Davidson, Mattias Merlier, Karel Bruneel, and Dirk Stroobandt. 2012b. A dynamically reconfigurable pattern matcher for regular expressions on FPGA. In *Advances in Parallel Computing*, Koen De Bosschere (Ed.), Vol. 22. IOS Press, Amsterdam, The Netherlands, 611–618.

Andre Maurice DeHon. 1996. *Reconfigurable Architectures for General-Purpose Computing*. AI Tech. Report 1586. MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Evelyn Duesterwald and Vasanth Bala. 2000. Software profiling for hot path prediction: Less is more. *SIGOPS Oper. Syst. Rev.* 34, 5 (Nov. 2000), 202–211. DOI:http://dx.doi.org/10.1145/384264.379241

Mentor Graphics. 2008. ModelSim® Reference Manual Software Version 6.5 g.

Scott Hauck and André Dehon. 2007. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann.

HES Group, Ghent Univeristy. 2012. TMAP Toolflow. Retrieved from https://github.com/UGent-HES/tlut_flow.

Karel Heyse, Brahim Al Farisi, Karel Bruneel, and Dirk Stroobandt. 2012. *Automating Reconfiguration Chain Generation for SRL-Based Run-Time Reconfiguration*. In Lecture Notes in Computer Science, Vol. 7199. Springer, Berlin, 1–12.

Nand Kumar, et al. 1995. Profile-driven behavioral synthesis for low-power VLSI systems. *IEEE Design & Test of Computers* 12.3 (1995), 70–84.

Srinivas Katkoori and Ranga Vemuri. 1998. Architectural power estimation based on behavior level profiling. *VLSI Des.* 7, 3 (1998), 255–270. DOI:http://dx.doi.org/10.1155/1998/93106

Eric R. Keller. 2000. Dynamic circuit specialization of a CORDIC processor. *Proc. SPIE* 4212 (2000), 134–141. DOI:http://dx.doi.org/10.1117/12.402517

Davin Lim and Mike Peattie. 2002. *Two flows for partial reconfiguration: Module based or small bit manipulations.* Xilinx Application Note 290 (v1.0).

Ming Liu, W. Kuehn, Zhonghai Lu, and A. Jantsch. 2009. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *International Conference on Field Programmable Logic and Applications, 2009 (FPL'09).* 498–502. DOI:http://dx.doi.org/10.1109/FPL.2009.5272463

Michael G. Lorenz, Luis Mengibar, Mario G. Valderas, and Luis Entrena. 2004. Power consumption reduction through dynamic reconfiguration. In *Field Programmable Logic and Application*, Jürgen Becker, Marco Platzner, and Serge Vernalde (Eds.). Lecture Notes in Computer Science, Vol. 3203. Springer, Berlin, 751–760. DOI:http://dx.doi.org/10.1007/978-3-540-30117-2_76

Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. 2003. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *SIGARCH Comput. Archit. News* 31, 2 (May 2003), 14–27. DOI:http://dx.doi.org/10.1145/871656.859621

Philippe Manet, Daniel Maufroid, Leonardo Tosi, Gregory Gailliard, Olivier Mulertt, Marco Di Ciano, Jean-Didier Legat, Denis Aulagnier, Christian Gamrat, Raffaele Liberati, et al. 2008. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP J. Embedded Syst.* 2008 (2008), 1.

opencores. 2012a. Pipelined AES. Retrieved from http://opencores.org/project,aes_pipe.

opencores. 2012b. RC6 cryptography. Retrieved from http://opencores.org/project,cryptography.

opencores. 2012c. Twofish. Retrieved from http://opencores.org/project,twofish.

Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. 2011. KLIMAX: Profiling memory write patterns to detect keystroke-harvesting malware. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Lecture Notes in Computer Science, Vol. 6961. Springer, Berlin, 81–100. DOI:http://dx.doi.org/10.1007/978-3-642-23644-0_5

Martin Roesch. 1999. Snort—Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA'99)*. USENIX Association, Berkeley, CA, 229–238.

Michael J. Wirthlin and Brad L. Hutchings. 1998. Improving functional density using run-time circuit reconfiguration FPGAs. *IEEE Transactions on Very Large Scale Integr. (VLSI) Syst.* 6, 2 (June 1998), 247–256. DOI:http://dx.doi.org/10.1109/92.678880

Michael J. Wirthlin and Brad L. Hutchings. 1997. Improving functional density through run-time constant propagation. In *Proceedings of the 1997 ACM 5th International Symposium on Field-Programmable Gate Arrays (FPGA'97)*. ACM, New York, NY, 86–92. DOI:http://dx.doi.org/10.1145/258305.258316

Inc. Xilinx. 2004. OPB HWICAP, ds 280.

Inc. Xilinx. 2006. OPB HWICAP (v1.00b), ds 280.

Inc. Xilinx. 2007. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide UG012 (v4.2)*. (2007).

Inc. Xilinx. 2011. LogiCORE IP XPS HWICAP (v5.01a) DS586. (2011).

Inc. Xilinx. 2012a. EDK concepts, tools, and techniques UG683. (2012).

Inc. Xilinx. 2012b. LogiCORE IP AXI HWICAP (v2.02a) DS817. (2012).

Inc. Xilinx. 2012c. *Synthesis and Simulation Design Guide UG626*. *(v13.4)* (2012).

Inc. Xilinx. 2012d. *Virtex-5 FPGA Configuration User Guide UG191 (v3.11)*. (2012).

Received December 2013; revised February 2014; accepted April 2014